ICT-257422

CHANGE

CHANGE: Enabling Innovation in the Internet Architecture through Flexible Flow-Processing Extensions

Specific Targeted Research Project

FP7 ICT Objective 1.1 The Network of the Future

D3.3: Flow Processing Platform: Main Implementation

Due date of deliverable: 30 September 2012

Actual submission date: October 26th, 2012

Start date of project	1 October 2010
Duration	36 months
Lead contractor for this deliverable	NEC
Version	Final, October 26, 2012
Confidentiality status	Public

© CHANGE Consortium 2012

Abstract

This deliverable describes the main implementation of the CHANGE flow processing platform and the source code distribution that accompanies this document, which is restricted. We begin by covering the platform requirements and the resulting platform design. We continue by describing the platform's API, that is, the set of functions that are externally visible. This allows CHANGE platforms to vary significantly from each other in terms of hardware while still being able to inter-operate. In addition, we provide a detailed description of the internal platform controller architecture. Finally, this document gives a brief description of how to install and run a CHANGE platform.

Target Audience

Network experts and researchers. This document is public, but the accompanying source code is restricted to the Commission Services and groups specified by the consortium.

Disclaimer

This document contains material, which is the copyright of certain CHANGE consortium parties, and may not be reproduced or copied without permission. All CHANGE consortium parties have agreed to the full publication of this document. The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the CHANGE consortium as a whole, nor a certain party of the CHANGE consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

This document does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of its content.

Impressum

Full project title	CHANGE: Enabling Innovation in the Internet Architecture through Flexible Flow-Processing Extensions
Title of the workpackage	Work Package 3. Flow Processing Platform Design and Development
Editor	Felipe Huici, NEC
Project Co-ordinator	Adam Kapovits, Eurescom
Technical Manager	Felipe Huici, NEC
Copyright notice	© 2012 Participants in project CHANGE

List of Authors

Authors	Felipe Huici (NEC), Costin Raiciu (PUB)
Participants	NEC (Editor), PUB
Work-package	WP3: Flow Processing Platform Design and Development
Confidentiality	Public (PU)
Nature	Report (R)
Version	1.0
Total number of pages	20

© CHANGE Consortium 2012

Contents

List of Authors 3

List of Figures 5

List of Tables 6

1 Introduction 7

1.1 Platform Requirements . . . 7

1.2 Flow Processing Platform Overview . . . 8

2 External Interface: Primitives 10

2.0.1 High-Level API . . . 10

2.0.2 Low-Level API . . . 11

2.0.3 Return Values . . . 11

2.0.4 Allocation Request Format . . . 12

3 Internal Architecture: the Controller 13

3.1 Controller Components . . . 13

3.2 Daemon Architecture and Communications . . . 14

3.3 Tasks and Commands . . . 14

3.3.1 Structure of Tasks and Commands . . . 15

3.3.2 Task Priorities . . . 15

3.3.3 Hot Patching . . . 15

4 Initial Software Implementation 16

4.0.4 Platform Hardware Description . . . 16

4.0.5 Platform Software Description . . . 16

4.0.6 Brief User's Manual . . . 17

4.0.6.1 Requirements . . . 17

4.0.6.2 Installation . . . 17

4.0.6.3 Running the Platform . . . 17

4.0.7 Wide-Area Network Testbed . . . 18

5 Conclusions 20

References 20

© CHANGE Consortium 2012

List of Figures

1.1 Flowstream Platform Overview. 8

3.1 Platform controller architecture overview, only control-plane connections are shown. 13

© CHANGE Consortium 2012

List of Tables

© CHANGE Consortium 2012

1 Introduction

The Internet has grown over the last twenty years to the point where it plays a crucial role in todays society and business. By almost every measure, the Internet is a great success. It interconnects over a billion people, running a wide range of applications, with new ones appearing regularly that take the world by storm. Yet despite this success the Internet comes with important shortcomings. The limitations are well-known: the Internet does not provide predictable quality of service, and does not provide a sufficiently robust and secure infrastructure for critical applications. Worse, making changes to the basic Internet infrastructure is costly, time-consuming and often unfeasible: operators are paid to run stable, always available networks which is anathema to deploying new mechanisms.

To overcome these problems CHANGE introduces a new evolutionary (i.e., incrementally deployable) architecture based around the notion of *flow processing platforms*, or *Flowstream platforms* for short, located throughout the network. The idea is that for specific flows that need special processing, flow owners discover platforms along the path and request flow processing from them. What do we mean by "flow processing"? Flow processing can encompass a large number of actions, from filtering, NAT, DPI to packet scrubbing and monitoring, among others. In fact, as we envision at least certain versions of these platforms to be based on general-purpose hardware, flow processing will be largely user-defined, and we expect the more interesting uses of platforms will be for things that are currently not in existence.

The core of the CHANGE architecture relies, then, on these Flowstream platforms. Other deliverables of the project (D4.2-D4.4) cover the architecture itself, including how users discover these platforms or how off-path packets can actually make it to a platform. This deliverable, on the other hand, focuses on describing the platforms themselves. We begin by giving a set of requirements for the platform, and give an overview of the basic hardware structure of a platform. From there we describe the controller itself, that is, the software in charge of managing the platform. We conclude with a description of the software distribution that accompanies this document, including a (brief) manual on how to set-up and run a CHANGE platform.

1.1 Platform Requirements

The goals outlined above dictate a number of requirements for the Flowstream platforms:

- **Flexibility**: platforms should be able to perform a wide range of flow processing.
- **Dynamic Installation**: platforms should be able to install new types of processing on demand.
- **Dynamic Scalability**: platforms should be able to scale their capabilities to meet dynamic demand.
- **Isolation**: platforms should be able to concurrently host different kinds of processing for different users without one user's processing affecting another's. This includes taking measures so that untrusted code is not able to adversely affect other users or the basic functionality of the platform.

- **Flow Statistics**: platforms should be able to provide at least basic on-demand per-flow statistics to requesting users.

- **High Performance**: the platform should perform flow processing with good performance.

1.2 Flow Processing Platform Overview

As mentioned, Flowstream platforms have to have a fair amount of flexibility, both in terms of processing and in the ability to dynamically install processing. While the CHANGE architecture does not dictate any particular implementation of the platforms as long as they conform to a set of APIs, recent studies have shown that x86 servers not only have the processing flexibility needed but can also yield high performance [**?**][**?**][**?**]. These, coupled with programmable commodity hardware switches such as OpenFlow switches and virtualization technologies such as XEN [**?**], form the basis of one type of Flowstream platform (see figure 1.1).

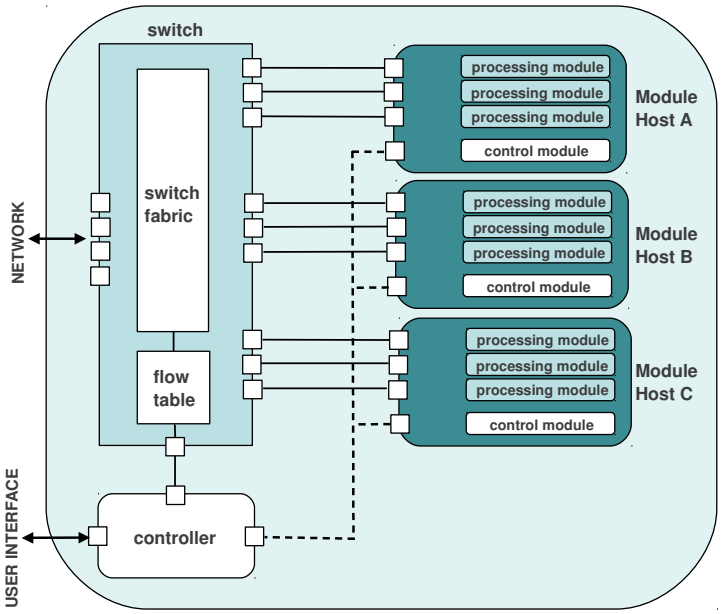

Figure 1.1: Flowstream Platform Overview.

In a Flowstream platform, a programmable switch such as an OpenFlow switch is used to distribute incoming flows to various *module hosts* (essentially x86 servers). The platform's capabilities can thus dynamically be scaled up by adding servers and down by shutting them down. Generally we assume module hosts to be computers containing general-purpose CPUs (e.g., x86 or x86_64 architectures). What system they actually run can vary. For instance, a module host could run XEN, another one Linux and yet another one FreeBSD. In addition, the module hosts contain a number of entities called *processing modules* (PMs) where the actual network processing takes place. Here again, there are choices regarding a PM's implementation. These can range from a process in a Linux system (e.g., a process running Bro or Click [**?**]), a minimalistic OS or a full virtual machine running on XEN[1].

[1]The XEN case provides an easy way to provide isolation for untrusted code.

© CHANGE Consortium 2012

It is also worth pointing out that if needed, it is entirely possible to include a specialized hardware device (e.g., a DPI box) in a Flowstream platform. After the flows are processed by the processing modules, the switch is once again used to send them back out of the platform. Naturally, it might not always be necessary to send flows out; for instance, if Flowstream is being used to monitor mirrored flows the processing modules will act as sinks.

Flowstream also contains a controller which manages the entire platform and we will dedicate the rest of this document to its description.

2 External Interface: Primitives

CHANGE platforms can vary widely in terms of the hardware they contain, from large deployments in data-centers consisting of commodity servers connected via switches (e.g., Openflow), to blade servers and even single-host platforms. In order for such vastly different platforms to be able to interact under a common CHANGE architecture we need to abstract a common API that summarizes the primitive functionality that is needed in order to instantiate flow processing.

We've split the API into two parts: a high-level API which should be the most commonly used one; and a low-level API for more advanced uses of CHANGE platforms. The software distribution that accompanies this document implements both, but currently only makes the high-level API externally visible.

2.0.1 High-Level API

Here's a list of the functions that a CHANGE platform must implement when conforming to its high-level API:

- `install_allocation`: Installs an allocation on a CHANGE platform. This includes instantiating all necessary processing modules and creating connections between them and traffic coming into and out of the platform. Returns an id for the allocation.

Parameters		
Name	**Type**	**Description**
`alloc`	`string`	The allocation request (see section below)

- `delete_allocation`: Deletes an allocation on a CHANGE platform. This includes removing processing modules and the connections between them. No return value.

Parameters		
Name	**Type**	**Description**
`alloc id`	`integer`	The id of the allocation to delete

- `deploy_pm`: Installs a new type of processing module on a CHANGE platform. This allows for remotely updating a platform. No return value.

Parameters		
Name	**Type**	**Description**
`pm_name`	`string`	The name of the new processing module
`pm_src`	`string`	The source code for the pm (see section 3).

- `get_pm_info`: Gets information about all processing modules currently supported by a platform. Returns a list of two-tuples. Each tuple has a string providing a human description of what the pm does and what parameter it takes, plus a dictionary describing the pm's primitives (e.g., "Read", see D2.4 section 4.4 for a full listing of these). Takes no parameters.

© CHANGE Consortium 2012

2.0.2 Low-Level API

Here's a list of the functions that a CHANGE platform must implement when conforming to its low-level API:

- `create_pm`: Instantiates a processing module on a CHANGE platform. No return value.

Parameters		
Name	**Type**	**Description**
`pm`	`PM`	The processing module to instantiate.

- `delete_pm`: Removes a processing module from a CHANGE platform. No return value.

Parameters		
Name	**Type**	**Description**
`pm`	`PM`	The processing module to remove.

- `create_connections`: Creates connections between processing modules on a platform. Currently this assumes the existence of an Openflow software switch. No return value.

Parameters		
Name	**Type**	**Description**
`switch_name`	`string`	The name of the Openflow switch.
`switch_table_id`	`integer`	The id of the table within the switch.
`alloc`	`string`	The allocation request describing the connections.

- `delete_connections`: Creates connections between processing modules on a platform. Currently this assumes the existence of an Openflow software switch. No return value.

Parameters		
Name	**Type**	**Description**
`switch_name`	`string`	The name of the Openflow switch.
`switch_table_id`	`integer`	The id of the table within the switch.
`alloc`	`string`	The allocation request describing the connections.

2.0.3 Return Values

All functions above return a common `ReturnValue` structure consisting of (1) a status code, (2) a message in the form of a string and (3) a function-specific return value. For the status code, the values supported are:

Code	Value	Explanation
CODE_NOT_READY	-1	The instruction is being carried out but is not finished
CODE_SUCCESS	0	The instruction was carried out successfully
CODE_FAILURE	1	The instruction encountered problems when executing

In the case of CODE_FAILURE, the message string that's part of `ReturnValue` should include an explanation of what the problem was. Note that functions in the API saying that they have no return value still return a `ReturnValue` object but with its third item set to void.

2.0.4 Allocation Request Format

At its most basic, a request to do network processing on a CHANGE platform (a *allocation request*) needs to specify three things: (1) a set of parametrized processing modules to process the actual packets; (2) how these processing modules should be inter-connected; and (3) where to receive packets from and where to send them to when finished.

To achieve this, CHANGE platforms use a very simple language to describe an allocation request[1]; here's an example:

```
BASESYS=xen.clickos


ft :: FromNet("tun0", "ip,nw_src=10.10.1.2,nw_dst=8.8.8.8")
tt :: ToNet("tun1")
mirror1 :: Mirror(")


ft -> [0]mirror1[0]
   -> tt
```

The first section of a config file defines key-value pairs. In the example, the configuration file that the processing module types should be instantiated as ClickOS virtual machines (a processing module such as a firewall may support different kinds of implementations, for instance as a ClickOS vm, a Linux vm, or perhaps an ipfw rule).

Next, the config defines the set of processing modules to be used, giving parameters for each of them where appropriate. Finally, the config specifies how the processing modules are connected. Each module can have a number of input and output ports, so users must explicitly specify which ports should be connected (the numbers in brackets in the example above).

Note that other related mechanisms such as authentication and access control are handled separately from the actual installation of an allocation request and are described in other deliverables.

[1] Note that in previous deliverables we were using an XML-based description language. In contrast, this new language is much simpler and accessible to users of the platform.

© CHANGE Consortium 2012

3 Internal Architecture: the Controller

Having described what a CHANGE platform looks like to the outside world, we now move on to describing its internals, and in particular the architecture of the controller software that manages the platform.

3.1 Controller Components

While logically the controller is a single entity, in actuality it is implemented as a set of inter-communicating daemons (see figure 3.1; only one module host is shown for simplicity's sake). The separation of the controller into a set of daemons allows us to break down its functionality into more manageable pieces that use common daemon code. Further, this allows for different implementations of the daemons to exist as long as they comply with a common interface. For instance, the Openflow daemon could have two versions: one for a Linux-based Openflow switch talking to another daemon on the switch, and another one using NOX to program a hardware switch using the Openflow protocol.

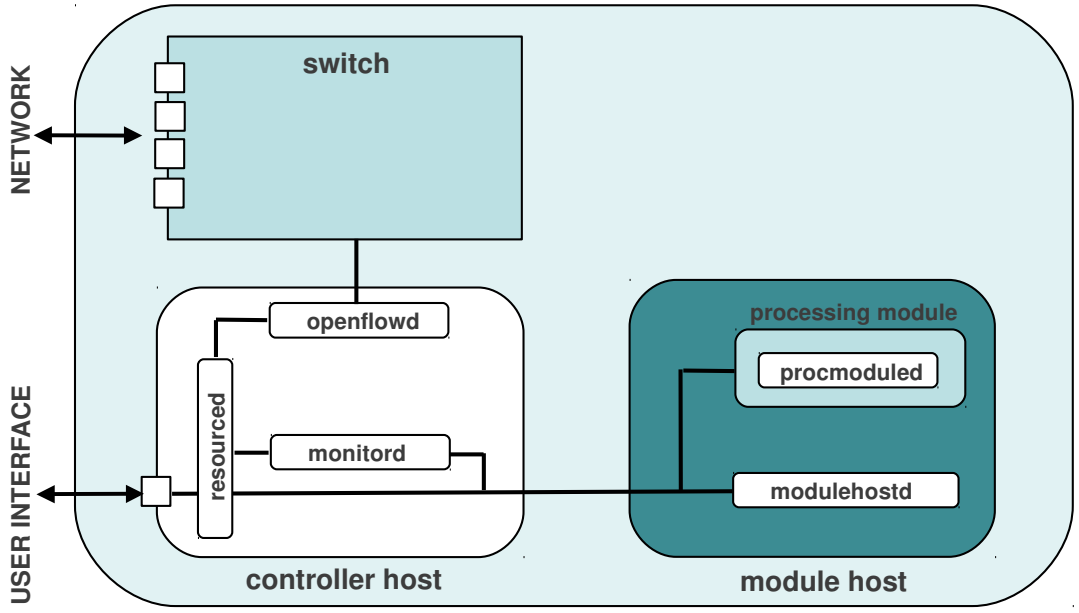

Figure 3.1: Platform controller architecture overview, only control-plane connections are shown.

The daemons comprising the Flowstream platform's controller are as follows:

- **Resource daemon**: This is the controller's main daemon, providing the interface to the outside world through which Flowstream users/clients submit *allocation requests* (as described in the previous chapter).

  This daemon takes care of receiving such requests and deciding how to allocate resources to them. It talks to the module host, processing module and Openflow daemons in order to install these allocations. In addition, it receives both the Openflow statistics from the Openflow daemon and performance statistics from the module hosts and processing modules via the monitoring daemon. Both of these are used as input to the resource allocation algorithms.

- **Openflow daemon**: This daemon is in charge of inserting and removing the Openflow entries that will direct traffic from the outward-facing ports (labeled "network" in the figure) to the necessary module hosts and processing modules and then back out. In addition, this daemon periodically retrieves Openflow statistics (e.g., per-flow byte and packet counts) and gives them to the resource daemon to use as input for its resource allocation algorithms. This daemon can make use of existing Openflow protocol implementations such as NOX [**?**] for communicating with the switch.

- **Module host daemon**: Runs on a module host and takes care, among other things, of instantiating and deleting processing modules, as well as setting up the necessary networking to get flows in and out of them. It also provides performance figures to the monitoring daemon about the current load of the module host.

- **Processing module daemon**: Runs on a processing module and handles modifications to the processing it is doing, as well as reporting performance statistics to the monitoring daemon. Note that depending on the implementation of the module host, it may or not be possible to run a daemon directly on the processing module. For instance, a processing module implemented as a minimalistic OS running on a Xen module host may not be able to run the processing daemon. This could be overcome by running the daemon on the module host (i.e., Xen's dom0) and using Xen's interface to modify the properties of the processing module.

- **Monitoring daemon**: This daemon periodically gathers performance statistics about the platform from the module host daemons, processing module daemons and the switch daemon, and gives these to the resource daemon, which uses the data as input to the resource allocation algorithms.

3.2 Daemon Architecture and Communications

At a high level, a daemon consists of a stand-alone process running an XML-RPC server. We chose XML-RPC since it is simple to use and widely available in different programming languages. A daemon takes care of executing two types of items: a *task* or a *command*. All daemons inherit from a common daemon class. The basic idea behind the architecture of this class is that most of a daemon's functionality resides in tasks and commands, while the daemon itself simply takes care of receiving requests and instantiating the right task or command for them. This simplifies the daemon itself, encourages code re-use (since commands or tasks can be shared by multiple daemons) and enables mechanisms such as hot-patching. The exception are functions that are common to many commands or tasks of the daemon; these functions typically reside in the class that implements the daemon, or alternatively could be put in a library file.

3.3 Tasks and Commands

The main difference between a task and a command is that a command will block while waiting for a result and a task will not. As a result, a command is generally meant to be used for carrying out a simple and

 © CHANGE Consortium 2012

quick operation, such as registering a module host with the resource daemon. A task, on the other hand, is asynchronous, and will immediately return after being called. Despite this, tasks, if so desired, can still return a value.

The final difference between tasks and commands is that tasks can be assigned normal or high priority, and also allow for compound tasks. A compound task is a task containing multiple sequential tasks. Tasks within a compound task are executed by the daemon in the same order as they were added to the compound tasks (FIFO). As a result, a compound task provides a convenient way of doing sequential execution of multiple tasks. Of course, a developer could instead choose to mimic the mechanism using multiple regular tasks, but the compound task interface is cleaner.

3.3.1 Structure of Tasks and Commands

The Flowstream controller's tasks and commands will follow a two-tier structure. Tasks and commands common to all daemons (for instance, a task to upgrade a daemon's set of commands and tasks) will reside in a common directory, while additional per-daemon directories will contain daemon-specific functionality.

In order to implement a new task (and similarly for a command), developers place a file in the relevant daemon's sub-directory under tasks, and in that file place a class that inherits from a task super-class. The subclass will have to implement a `run_task` method that receives a single parameter containing all of the method's parameters. Which data structure this single parameter is depends entirely on the command's implementer. If the method requires more than one parameter, these need to be packed into the single parameter, for instance by using a list. After the method finishes performing its function, it returns a common return type to signal the result of the operation back to the caller.

3.3.2 Task Priorities

Tasks will have at least two priority types, high and normal. As their names indicate, these are used to give priority to some tasks over others, with normal tasks only running when no high priority tasks exist at the daemon. For example, tasks that carry out background monitoring (for example, checking CPU loads on module hosts) would be invoked with normal priority, while installing a user configuration would use high priority.

3.3.3 Hot Patching

The controller's daemons include a hot patching mechanism in order to provide the ability to upgrade a daemon's functionality without having to restart it. The simplest way of achieving this is to have all tasks and commands dynamically loaded. In other words, when a request to execute a task arrives at a daemon, the daemon looks in the relevant directory to see if a task with the given name exists, and if it does, loads the task's code and runs it. With this in place, introducing new tasks or commands to a daemon simplifies to putting the relevant files in the daemon's directory. For tasks or commands that are already loaded, their respective modules are reloaded.

© CHANGE Consortium 2012

4 Initial Software Implementation

Previous chapters described the requirements and design of the platform, both from an external and internal point of view. We now dedicate the rest of this document to describing the actual implementation of the platform that is provided in the software distribution that accompanies this deliverable.

4.0.4 Platform Hardware Description

As previously mentioned, a platform's hardware can consist of an entire data-center, a single rack of x86 servers connected via programmable switches or something smaller like a blade server. In essence, the software abstracts from this by only assuming the existence of some sort of programmable switch (hardware or software) that can pipe together processing modules sitting on module hosts. The only other assumption is the availability of a server able to execute Python, the programming language that the platform controller is coded in.

While the software can support different kinds of platforms, so far we have focused on *single-host* platforms, that is, an entire platform implemented on a single x86 server. The main reason behind this is that it simplifies the deployment story when trying to set-up the CHANGE wide-area testbed (see section at the end of this chapter) since it would be hard to have partners make an entire rack of computers available. Having said that, it is likely that at least for the boot camp there will be a larger version of the platform (i.e., with several servers) available.

The platform consists, then, of a single x86 server running the controller software. It has a single module host (itself), and uses Openvswitch [**?**], a software programmable switch that implements the Openflow protocol among others, to pipe traffic between the platform's external interfaces and the processing modules it runs.

For this first version of the platform we have chosen ClickOS as the system for implementing the processing modules. In essence, the x86 server runs Xen, and the platform controller runs in Xen's dom0, while the processing modules are each a Xen virtual machine. Further, each processing module has (at least one) virtual interface that gets added to Openvswitch when the virtual machine is created. Again, the platform's software is not bound to only ClickOS, and it is possible that in the future we will support other systems like FlowOS for instantiating processing modules.

4.0.5 Platform Software Description

The distribution contains a tree structure consisting of the following directories and files:

- `cmds`: The commands for the various controller daemons.

- `configs`: Contains three types of configuration files: (1) platform description files describing a platform's hardware; (2) base system description files describing the software/OS running on a platform; and (3) allocation description files, samples of the type of processing that can be instantiated on a platform.

© CHANGE Consortium 2012

- `core`: The main files of the platform controller.
- `daemons`: The platform daemons.
- `pms`: The various processing modules that a platform supports go here.
- `tasks`: The tasks for the various controller daemons.

In addition, the distribution comes with the file `flowstream_start.py` that takes care of starting the platform's controller (i.e., all of its daemons) and the file `flowstream_exec.py` which allows for the installation of an allocation request (i.e., network processing on the platform). Usage for this is described in the next section.

4.0.6 Brief User's Manual

In this section we give a brief explanation of what's required to run a CHANGE platform, including how to install its software. We further cover how to get the controller running and how to install and delete allocation requests (i.e., network processing) from it.

4.0.6.1 Requirements

The platform controller is implemented in Python, so a somewhat recent version of it is needed (at least version 2.6). The platform's current version uses ClickOS, for which Xen 4.1.2 is required. Further, the software expects an Openvswitch switch to be in Xen's dom0, and for Xen's networking scripts to be adapted such that the virtual machines' virtual network interfaces are added to the switch. At least one physical interface (the one that the platform is supposed to receive packets from) should be added to this switch.

4.0.6.2 Installation

Begin by unpacking the distribution tar ball with:

```
tar -xvzf flowstream.tgz
```

Next, you'll need to modify a few things in the following platform configuration file:

```
configs/platformdesc/platdesc_singlehost_xen.xml
```

Mostly all you need to change are the paths to "xencfg" and "clickoslib" to point to the directories containing the ClickOS Xen configuration and front-end API files, respectively. You might also want to modify the directories that daemons log messages to (look at the `log_dir` tags).
With this in place, you're now ready to run the CHANGE platform.

4.0.6.3 Running the Platform

To start with, set the `PYTHONPATH` environment variable to point to the platform's code:

```
export PYTHONPATH=[path/to/platform/srcs]
```

The distribution comes with a script for starting all the necessary daemons:

```
usage: flowstream_start.py [description file] [(module_host=yes|no)]}
```

Make sure you run the process as root and that you use a full path for the description file (this is the file you modified in the previous section). Set the `module_host` parameter to "no" (this is to distinguish from a platform that contains multiple servers/module hosts). If successful, you should see console output stating that the daemons are running. Note that the daemons will log messages to the directories specified in the platform description file.

Once the controller is running, we're ready to ask the platform to instantiate network processing. The first step would be to create an allocation request configuration file (refer back to 2.0.4 and have a look at `mirrortest.conf` in the allocation configuration directory for examples). Once this is done, we can the following command to install it:

```
usage: python flowstream_exec.py [command] [(params)...]
```

For example, to install a configuration called "test.conf", we would run:

```
python flowstream_exec.py InstallAllocation test.conf
```

If successfull, the command will print out the id of the (now running) allocation. We can then later use this id to delete the allocation from the platform:

```
python flowstream_exec.py DeleteAllocation [allocation id]
```

4.0.7 Wide-Area Network Testbed

In order to test the CHANGE architecture in a realistic environment (e.g., one containing middleboxes, delays, jitter, etc) we have started to deploy CHANGE platforms at various partner sites. We had initially considered using Planetlab for this, as this would have given us a larger number of platforms and perhaps more geographical diversity. However, Planetlab's boxes are rather restrictive: they run very old distributions, making it hard to install some of the basic packages that our software rely on (even installing a somewhat recent version of Python is challenging!); they do not allow users to tinker with network interfaces (even virtual ones like tap devices); and clearly they cannot support a system like ClickOS which requires Xen. Still, the controller software includes support for running a "Planetlab platform" using user-level Click to instantiate processing modules in case it is needed in the future.

At this point in time we have set-up or are in the processing of setting up CHANGE platforms at the following partner locations:

- NEC
- PUB
- UCL-BE

© CHANGE Consortium 2012

- TUB

It is likely that in the future we will add other platforms.

5 Conclusions

In this deliverable we covered the requirements, design and main implementation of the CHANGE platform. In terms of hardware, a CHANGE platform can range from an entire data center, a rack of servers inter-connected by a programmable switch, a blade server, or to a single x86 server. The platform's controller software can potentially support all of these, exposing a common interface so that different types of CHANGE platforms can inter-operate.

This document further provided a detailed description of the controller as well as a brief user's manual depicting how to set up a platform and have network processing installed on it. So far the software distribution focuses on a single-server platform running ClickOS for its processing modules, but it would be relatively easy to extend this platform to setups containing more servers (as we might do for the CHANGE boot camp).

As future work, we're currently in the process of setting up a wide-area testbed composed of CHANGE platforms at (initially at least) four different geographical locations. The platform software is already running at one of these, and we are currently finishing the install at other locations with the goal of testing the CHANGE architecture as described in deliverable D4.4.

© CHANGE Consortium 2012